Relational Database Systems I

Christoph Lofi
Simon Barthel

Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de
**Views**

- Views create logical “tables” from a given query
  - Can be used, for example, for easier querying or for security reasons

- Can be materialized
  - May speed up some things, but also introduces issues with consistency

- Are usually read-only, but can be updatable
  - Only if certain requirements are fulfilled
• **Indexes**
  
  – **Used to locate tuples quicker**
    
    • Usually, will lead to faster queries
  
  – **Primary indexes**
    
    • Imply the physical order of tuples in the table with respect to the primary key attributes
  
  – **Secondary indexes**
    
    • Optional indexes for non-primary key attributes which create additional data structures
    
    • Beneficial for foreign key attributes if join queries are often used
• **Transactions**
  – Groups multiple database SQL statements into one functional unit
    • Default in many DBMS: Each single statement is a transaction
  – Important for **critical applications** and **multi-user environments**
    • Prevents anomalies and inconsistencies resulting from, e.g., database failures and concurrent accesses
  – Each group of statements is executed in a **controlled** fashion
    • **ACID** properties: Atomicity, Consistency, Isolation, Durability
    • “Transactions are either executed fully or not at all, and will not be affected by partial results of other database transactions”
12 Accessing Databases

• Database access using a library (application programming interface, API)
  – most popular approach
  – prominent examples
    • CLI (Call level interface)
    • ODBC (Open Database Connectivity)
    • JDBC (Java Database Connectivity)
General steps in using database APIs

- set up the environment
- define and establish connections to the DBMS
- create and execute statements (strings)
- process the results (using the cursor concept)
- close the connections
• Call Level Interface
• ODBC
• JDBC
  – usage
  – prepared statements
  – transactions
12.1 CLI

• The **Call Level Interface (CLI)** is an ISO software standard developed in the early 1990s
  – defines how programs send queries to DBMS and how result sets are returned
  – was originally targeted for C and Cobol

• **Vision:** **Common Application Environment**
  – set of standards and tools to develop open applications
  – allows to integrate different programming teams and DB vendors
• CLI libraries are provided by the DB vendors
  – each library is specific for the respective DBMS and follows the individual DBMS’s syntax
  – however, vendor libraries all follow the CLI standard and can be used interchangeably by all applications
• Host language connects and accesses DB using following concepts
  – **environments**: represent the DBMS installation
    • properties and specific settings & drivers
  – **connections**: a current session with the DBMS
    • URL, username, password, session context
  – **statements**: SQL statements to be passed to DBMS via a connection
  – **descriptions**: records about tuples from a query or parameters of a statement
    • number of attributes and respective types
    • parameters in function calls
• An environment can host several connections, while a connection can host several statements.
When working with CLI, **following steps** have to be performed

- include the CLI function libraries and open database connections
- metadata about the database, tables, and columns can be retrieved
- define variables to contain SQL statement information
- execute the query and manipulate the result set in a (implicitly declared) cursor
- terminate statements, connections and the environment
12.1 CLI: Handle Concept

- Function `SQLAllocHandle(T, I, O)` is used to create data structures (variables), which are called environment, connection, and statement handles
  - **T**: Handle type, e.g., an environment, a connection, or a statement
  - **I**: Input handle, container structure at next higher level (statement < connection < environment)
  - **O**: Output handle (pointer to new handle)

- Example for handling statements
  - `SQLAllocHandle(SQL_HANDLE_STMT, myCon, myStat);`
    - myCon is a previously created connection handle.
    - myStat is the name of the statement handle that will be created.
12.1 CLI: Handle Concept

• For details please read the manual... 😊

– example in C

```c
#include <sqlcli.h>
SQLRETURN ReturnCodeEnv;
SQLHENV EnvironmentHandle;
...
ReturnCodeEnv = SQLAllocHandle(
    SQL_HANDLE_ENV,
    SQL_NULL_HANDLE,
    &EnvironmentHandle
);
```
12.1 CLI Basic Control Flow

1. `AllocHandle(Statement)`
   - **Want repeatable execution?**
     - No
     - Specify dynamic arguments
       - `ExecDirect()`
     - Yes
     - Prepare()
       - Specify dynamic arguments
         - `Execute()`
   - `cursurspecification`
     - `GetDiagField()` to get stmt. type
     - DELETE, INSERT or UPDATE
   - Get number and type of columns
     - Bind column variables
       - `Fetch()` Process column data
         - Yes
           - More rows?
             - Yes
               - More statements?
                 - Yes
                   - More transactions?
                     - Yes
                       - Disconnect()
                         - FreeHandle(Connection)
                         - FreeHandle(Environment)
                 - No
                   - EndTran()
                     - Yes
                       - Disconnect()
                         - FreeHandle(Connection)
                         - FreeHandle(Environment)
                 - No
                   - Disconnect()
                     - FreeHandle(Connection)
                     - FreeHandle(Environment)
     - Others
       - `GetDiagField()` to get row count
         - No
           - Want to repeat statement?
             - No
               - `FreeHandle(Statement)`
             - Yes
               - `AllocHandle(Statement)`
The complete technical standard is available freely from the Open Group

Specification C451

Over 300 pages...

• Call Level Interface
• ODBC
• JDBC
  – usage
  – prepared statements
  – transactions
The Open Database Connectivity (ODBC) provides a standardized application programming interface to DBMS using the CLI standard

- development driven by Microsoft in 1992, later versions aligned with X/Open and ISO/IEC
- builds on several CLI specifications, but does not implement full SQL features
- central for the design was independence of programming language, operating system, and DBMS
- implements the standardized middleware concept
• Basic idea: The DBMS is virtualized

– the person with specialized knowledge to make the application logic work with the database is the driver developer and not the application programmer

– application developers write to a generic DBMS interface and loadable drivers map that logic to vendor-specific commands
12.2 ODBC

- Being a **middleware solution** a basic implementation of ODBC always contains...
  - a generic **ODBC driver manager library** to interpret the applications’ commands
    - defines standard types and features
  - and a set of **database drivers** to provide the DBMS-specific details
    - each database vendors can write an individual driver to map ODBC commands
• ODBC supports different **numbers of tiers** that have to be passed to access the databases

  – Tier 1
    • **direct access** to database files by the database drivers (usually only used for desktop databases)

  – Tier 2
    • the database drivers prepare the requests and **pass them on to the DBMS** for execution (which is the normal case)

  – Tier 3
    • the database drivers prepare the requests and pass them to a specific **ODBC gateway** that manages the communication to the DBMS (e.g., via a low level interface) for execution
• ODBC development has driven by the need of easy application programming
  – originally in Microsoft’s Visual Basic
  – but quickly adapted for use in C, C++ and other languages

• The ODBC architecture also has certain drawbacks
  – large client networks may need a variety of drivers increasing the system-administration overhead
  – multi-tier ODBC drivers can ease this problem
• ODBC uses standard CLI calls…

• The concept of handles is used to set up the environment and connections
  – first, applications have to allocate an environment handle by calling SQLAllocEnv
  – then, a handle for a database connection (SQLAllocConnect) has to be allocated before calling connection functions like SQLConnect
To process SQL statements...

- an application must first acquire a **statement handle** by calling `SQLAllocStmt`
- there is a function for **direct execution** of a SQL statement (`SQLExecDirect`) and functions to **prepare** and **execute statements** (`SQLPrepare` and `SQLExecute`)
- an application can use **named cursors** by getting and setting the cursor name for a statement (`SQLGetCursorName` and `SQLSetCursorName`)
- an application **retrieves a row** in a result set by calling `SQLFetch`
- ...

Relational Database Systems 1 – Christoph Lofi – Institut für Informationssysteme – TU Braunschweig
• As part of ODBC’s termination logic…
  
  – every application should free statement handles using SQLFreeStmt\_close
  
  – close the connection and free the connection and environment handles by calling SQL\_Disconnect, SQL\_FreeConnec\_t, and SQL\_FreeEnv
  
  – we won’t go into more implementation details here, but consider the exact use for the case of JDBC
• Call Level Interface
• ODBC
• JDBC
  – usage
  – prepared statements
  – transactions
JDBC provides a standard Java library for accessing tabular data

- tabular data usually means a relational DBMS
- API provides standard way to connect to a DB
- API allows to perform dynamic queries
- method to create stored (parameterized) queries
- provides some (limited) data types for Java/DB impedance mismatch
  - result sets with rows and columns
  - methods for accessing table meta data
- provides functionality independent of chosen DBMS
12.3 JDBC

• JDBC does **not standardize SQL**
  – SQL statements are treated as **Java strings**
  – in case of full dynamic SQL, sometimes excessive **string manipulation** is necessary
  – if DBMS uses different/extended SQL syntax, this has to be considered by the programmer

• JDBC is not an acronym, but a registered product **trademark** by Oracle (used to be Sun Micros.)
  – however, usually, it is assumed that it stands for **Java Database Connectivity**
12.3 JDBC

Why not just use ODBC?

- ODBC is based on **binary libraries** (usually written in C)
  - native calls necessary
  - not platform-independent which is one of Java’s goals
  - ODBC drivers are often registered in the OS

- 1:1 translation from ODBC to Java does not work as ODBC heavily **relies on pointers**

- ODBC API is more **complex** and **littered** (and thus harder to learn and use)
  - for example, programmer needs to worry about byte alignment and advanced connection properties explicitly

- intention was to create a “**pure**” **Java** and fully portable API
  - no installation required, JDBC can easily be bundled into the application archive
12.3 JDBC

• JDBC is composed of **two primary components**

• **JDBC API:** An *programming interface* for database connectivity.
  
  – written in **100% pure Java**
  
  – completely **independent** of platform, vendor, and DBMS
  
  – provided by the Sun in its **Java SDK by default**
    
    • usually to be found in `java.sql` and `javax.sql`
12.3 JDBC

• JDBC driver
  – implementation of the respective API interface, responsible for communication with the database
  – interface implementation in Java, but may depend on any amount of binary libraries, middleware, or other tools
  – heavily dependent on the used DBMS
  – usually provided by the DB vendor
12.3 JDBC

• General Architecture
  – java application uses API
  – API uses driver
  – driver communicates with DB

• If you change the DBMS, you need to
  – provide a new driver
  – change configuration of driver
  – assuming the SQL syntax is compatible, you are done
    • if not, you are in trouble…
12.3 JDBC: Versions

- There are several versions of JDBC, each with improved functionality

<table>
<thead>
<tr>
<th>Version</th>
<th>Year</th>
<th>Java Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC 4.2</td>
<td>2013</td>
<td>Java 8</td>
</tr>
<tr>
<td>JDBC 4.1</td>
<td>2011</td>
<td>Java 7</td>
</tr>
<tr>
<td>JDBC 4.0</td>
<td>2006</td>
<td>Java 6</td>
</tr>
<tr>
<td>JDBC 3.0</td>
<td>2001</td>
<td>Java 1.4 &amp; Java 5</td>
</tr>
<tr>
<td>JDBC 2.1</td>
<td>1999</td>
<td>Java 1.3</td>
</tr>
<tr>
<td>JDBC 1.2</td>
<td>1997</td>
<td>Java 1.1</td>
</tr>
</tbody>
</table>

- JDBC drivers are written for a specific JDBC version
  - driver should match the JDBC version
  - however, most features also work with outdated drivers

- JDBC 3.0 is still very common!
  - However, 4.0 provides many major and valuable improvements!
• Beside versions, there are JDBC levels
  – comparable to ODBC tiers
  – for each level, there are different drivers
    • be careful when picking your driver! You need the right version and correct level!
  – all levels offer the same functionality (i.e., API is the same), but use different means of driver implementation and communication with the DBMS
    • different performance and portability properties
12.3 JDBC: Levels

- **Level 1: JDBC/ODBC bridge**
  - JDBC driver just translates requests to ODBC calls
    - **performance overhead** due to translation
  - needs correctly installed ODBC drivers on every client machine
    - distribution difficult
    - ODBC drivers are not platform-independent
12.3 JDBC: Levels

- **Level 2: Native API**
  - JDBC driver uses native calls to connect to a proprietary client software which handles DB connections
    - e.g. ORACLE client (which is a 1.7 GB installation)
  - difficult to port and with deployment problems
  - often used as cheap-and-dirty solution for older systems
    - also, may be a good idea if application is running on the same machine as the DBMS
### 12.3 JDBC: Levels

**Level 3: Middleware**

- JDBC driver communicates with a **middleware** software instead of the DBMS
- Often used for **large-scale enterprise applications** in a multi-tier environment
- Vendor specific translation may happen at the middleware
  - Just one client driver for any used DBMS
  - Middleware encapsulates the actual DBMS
    - Useful for advanced clustering, extended security, logging, caching, pooling, etc..

![JDBC Level 3 (Middleware)](image)
12.3 JDBC - Levels

- **Level 4: Direct pure Java**
  - driver purely written in Java
    - no call translation
    - no installation, no deployment problems
    - full portability due to platform-independence
  - driver connects directly to the DBMS
    - you need a different driver for each different DBMS
    - superior performance in remote scenarios
    - for access to a local DBMS, Level 1 might be better
• Basic steps when working with JDBC

1. **Load** the driver
2. **Define** a connection URL
3. **Establish** a connection
4. **Create** a statement(s)
5. **Execute** a statement(s)
6. **Process** the result(s)
7. **Close** the connection
12.3 JDBC: Create a Connection

- The **connection** is necessary to interact with a DBMS Server
  - Established by a **Driver** provided by your DBMS vendor
  - The DBMS server is specified using a **URL**
    - **jdbc:**[**driverAlias**]:[**driverParameters**]
    - **DB2** Level 4: URL
      - `jdbc:db2://[server][:port]/[db-name]`
      - example: `jdbc:db2://myserver.de:50000/hero_db`
    - **SQLite** Level 4 URL:
      - `jdbc:sqlite:[filename]`
      - example: `jdbc:sqlite:test.db`
    - further URL formats for most DBMS
The **Driver** instances are normally not managed manually

- Instead use the **DriverManager**
  - **DriverManager.getConnection(String URL)** provides you with a **Connection** object
  - Automatically searches a suitable driver in the Classpath matching the URL
    - Driver implements an **acceptsURL(String URL)** method
  - In earlier JDBC Versions it was necessary to load and register Drivers using
    - **Class.forName("path.to.driver");**

```java
public Connection getConnection(Properties props) throws SQLException {
    return DriverManager.getConnection("jdbc:db2://dblab.ifis.cs.tu-bs.de:50000/DBLAB", props);
}
```
12.3 JDBC: Statements

• To actually execute an SQL statement you need a **Statement object**
  – Created by a **Connection object**
  – There are three types of Statements
    • **Statement**
      – SQL directly written in the Statement
      – To be executed once
    • **PreparedStatement**
      – Used for frequent statements
      – Statement is provided as **parameterized String**
      – For each execution, **parameters are replaced by values**
    • **CallableStatement**
      – Used to execute server-side stored procedure (UDF)
• Using simple `Statement` objects
  – create a `Statement object` with the connection
    • `conn.createStatement();`
  – call one of the three execution methods
    • `executeQuery` (String query):
      – use for `SELECT` queries
      – returns a `ResultSet`
    • `executeUpdate` (String query):
      – use for any DDL or data changing statements (INSERT, UPDATE, DELETE)
      – returns an integer with number of affected rows
    • `execute` (String query):
      – advanced feature for multi-`ResultSet` queries

```java
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT count(*) FROM IMDB.title");
```
To access a query result, JDBC provides a `ResultSet`

- rows are retrieved one after another from the server
  - inspired by (but not compatible to) the Java iterators
  - a `cursor` is held pointing the current row in the server-side result set
- at first, the result set points **before** the first row
  - so, initially to no row at all
- `next()` method moves the cursor to the next row
  - returns true, if there is a next row

```java
ResultSet rs = stmt.executeQuery("SELECT ...")
while(rs.next()) {
    // do something with the current row
}
```
To read the columns of a row, there are multiple getters

- named `getX()` (e.g. `getInt()`, `getDouble()`)
  - access columns by name or by number (starting at 1)
- there are getters for each data type
  - each SQL data type is mapped to a Java data type

```java
ResultSet rs = stmt.executeQuery("SELECT id, real_name FROM heroes");
while (rs.next()) {
    int id = rs.getInt(1);
    String realName = rs.getString("real_name");
    System.out.println(id + ":" + realName);
}
```
### 12.3 JDBC: Data Types

**Example: Extract from direct JDBC data types for DB2**

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>SQL Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, java.lang.Integer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long, java.lang.Long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>java.io.StringReader</td>
<td>CLOB(n)</td>
</tr>
<tr>
<td>byte[]</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>BLOB(n)</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME</td>
</tr>
</tbody>
</table>
• After the result set rows has been read, the statement is marked **complete**

  – Statements and Results are usually **garbage collected** by Java

  – however, it is highly recommended to **manually close** statements (and thus result sets)

• `stmt.close()`

• potentially saves system resources
• Default `ResultSet` only allows moving the cursor forward and is read-only

– can be manipulated by using **Navigation options:**

  • `TYPE_FORWARDONLY` (default) to allow only forward movement
  • `TYPE_SCROLL_INSENSITIVE` to allow forward, backward and random movement
    – `previous()`: moves cursor to the previous row
    – `beforeFirst()`: moves cursor before the first row
    – `relative(int x)`: moves cursor x rows forward (or backward if x is negative)
    – `absolute(int x)`: moves cursor to the given absolute row number
12.3 JDBC: Result Set Options

- **TYPE_SCROLL_SENSITIVE**
  - same as **TYPE_SCROLL_INSENSITIVE**
  - **but** changes to the underlying data are directly visible in the result (i.e. rows are always up-to-date)
  - may have bad performance

```java
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
```
12.3 JDBC: Result Set Options

- **Update** options
  - **CONCUR_READ_ONLY** (default)
    - results can only be read
    - unlimited concurrency
  - **CONCUR_UPDATABLE** to allow updates
    - use `updateX` methods to update the current row (similar to `getX` methods)
      - updates are performed after you call `updateRow()`
      - if you want to cancel the updates, call `cancelRowUpdates()`
      - if you move to another row without `updateRow()`, nothing happens
    - may degenerate performance in massively concurrent applications due to lock contention thrashing (see RDB2)
12.3 JDBC: Result Set Options

– can also be used to insert rows
  • use `moveToInsertRow()` to move to a special insert row
  • use update-methods to set the values for the new row
  • then call `insertRow()` to commit the insert (cursor returns to the previous position)

– or to delete rows
  • call `deleteRow()` to delete the current row

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = stmt.executeQuery("SELECT id, real_name FROM heroes");
while (rs.next()){
    rs.updateString("real_name",
        rs.getString("real_name").toUpperCase()
    );
    rs.updateRow();
}
```
12.3 JDBC: Result Set Options

```java
Statement stmt = conn.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE
);
ResultSet rs = stmt.executeQuery(
    "SELECT id, real_name FROM heroes"
);

rs.moveToInsertRow();
rs.updateInt(1, 999);
rs.updateString(2, "Peter Parker");
rs.insertRow();

while(rs.next()) {
    if(rs.getString("real_name").equals("Bruce Banner")) {
        rs.deleteRow();
    }
}
```
12.3 JDBC: Receive Metadata

- The database metadata can be accessed via a `DatabaseMetaData` object
  - can be received using the connection
    - using `conn.getMetaData();`
  - Metadata includes
    - DBMS name, version, installation properties
    - available `schemas`, `tables`, `columns`
    - `primary keys` for a given table
    - ...

```java
DatabaseMetaData metaData = conn.getMetaData();
String dbmsName = metaData.getDatabaseProductName();
ResultSet schemas = metaData.getSchemas();
```
When performing a simple statement, roughly the following happens:

- The statement is composed in your app using `String manipulation`.
- The SQL String is `wrapped` and sent to the database via the JDBC driver.
- The DBMS `parses` and `checks` the statement.
- The DBMS `compiles` the statement.
- The DBMS `optimizes` the statement and tries to find the best access path.
- The statement is `executed`.

When you execute the same/similar statement multiple times, all those steps are performed for each single statement.
To avoid unnecessary overhead, **prepared statements** may be used.

Prepared statements use parameterized SQL:

- Use '?' as markers for parameters.
- Example:
  - `SELECT * FROM heroes WHERE id = ?`

Generic SQL query for retrieving an hero by its ID.

Prepared Statements may either be used for queries or for updates / DDL operations.
• Prepared Statements use the following workflow
  – when creating a (parameterized) prepared statement, it is wrapped, sent to the DBMS, parsed, checked, and optimized
    • only once for any number of execution
  – each time it is executed, the values for the parameters are transferred to the DBMS and the statement is executed
  – performance may be significantly higher compared to using dynamic statements
12.3 JDBC: Prepared Statements

• To supply values for the placeholders, use `setX(number, value)` methods
  – like for the get and update methods, there are set methods for any data type
    • placeholders are referenced by the position in the SQL string starting with 1
  – After all placeholders are filled, you may call
    • `executeQuery()` for queries returning a `ResultSet`
    • `executeUpdate()` for update/DDL statements return the number of affected rows

```java
PreparedStatement moviesInYears = conn.prepareStatement("SELECT * FROM movies WHERE releaseDate=>? AND releaseDate=<?"");
for (int i=0; i<10; i++) {
    moviesInYears.setInt(1, 1990+i*2);
    moviesInYears.setInt(2, 1991+i*2);
    ResultSet rs = moviesInYears.executeQuery();
    // ... do something
}
```
12.3 JDBC: Transactions

• Of course, you can use transactions within JDBC
  – transactions are normally disabled by default (depending on the DBMS)
    • "auto-commit"-mode is normally active
  – use `setAutoCommit(boolean switch)` to change transactional behavior
    • `true`: Every statement is executed immediately
    • `false`: Statement execution is held back until `COMMIT` is called
When transactions are enabled, any number of statements is considered as one transaction until it is **committed** or **canceled**

- to **commit** a transaction use
  
  • `conn.commit()`

- you may also create **save points**
  
  • `conn.setSavepoint(String savepointName)`

- to **roll back** use
  
  • `conn.rollback()`
  
  • or `conn.rollback(String savepointName)` to return to a given safe point
conn.setAutoCommit(false);

PreparedStatement changeNameStmt = conn.prepareStatement("UPDATE hero SET name=? WHERE name=?");

changeNameStmt.setString(1, "Jean Grey-Summers");
changeNameStmt.setString(2, "Jean Grey");
changeNameStmt.executeUpdate();

changeNameStmt.setString(1, "Scott Grey-Summers");
changeNameStmt.setString(2, "Scott Summers");
changeNameStmt.executeUpdate();

conn.commit();
Despite many years of development, JDBC is still a little bit clumsy.

Especially, there is no major release yet integrating with Java 8’s Lambda expressions.

However, we can try to mimic such an integration using some external libraries.

Example: Using Apache Commons DbUtils.

Likely, future versions of JDBC will properly support SQL with Lambda’s!
12.4 Beyond JDBC

- Apache Commons DbUtils
  - [http://commons.apache.org/proper/commons-dbutils/](http://commons.apache.org/proper/commons-dbutils/)
  - Simple API providing additional DB features, especially JavaBeans mappers
    - Example: Create a properly typed object stream!

```java
public class Player {
    String fn;
    String ln;

    public Player(String fn, String ln) {
        this.fn = fn;
        this.ln = ln;
    }

    @Override public String toString() {
        return fn + " " + ln;
    }
}
```
public void streamingSQL(Connection conn) throws SQLException {
    // open an object array stream
    Stream<Object[]> listStream = new QueryRunner().query(conn,
        "SELECT firstname, lastname FROM players",
        new ArrayListHandler()
    ).stream();

    // map objects to a Player stream
    Stream<Player> playerStream = listStream.map(array ->
        new Player((String)array[0], (String)array[1])
    );

    // print players
    playerStream.forEach(System.out::println);
}
• General Problem of Object Persistence
• Manual persistence
  – Generating UIDs
• Persistence frameworks
  – JPA
• Object databases
  – db4o